Title: Parallelization of a Proxy Transport App Using ComputeCPP and SYCL

Author(s): Burke, Timothy Patrick

Intended for: Report

Issued: 2019-06-18

Parallelization of a Proxy Transport App Using ComputeCPP and SYCL

Timothy P. Burke

June 10, 2019

tpburke@lanl.gov

I Introduction

This document details the attempts to parallelize a multi-group Monte Carlo proxy app using SYCL [1] and the ComputeCPP Community Edition compiler [2]. SYCL is a standard set by the Khronos Group and is described as a, "cross-platform abstraction layer that builds on the underlying concepts, portability and efficiency of OpenCL that enables code for heterogeneous processors to be written in a "single-source" style using completely standard C++." ComputeCpp Community Edition (CE) is a heterogeneous parallel programming platform that provides a conformant implementation of the SYCL™ 1.2.1 Khronos specification. ComputeCPP provides a compiler and an SDK containing a build system to improve the usability of the compiler. The supported OpenCL 1.2 platforms for ComputeCpp are AMD® and Intel®, with experimental support for PTX which is used on Nvidia graphics cards as well as experimental support for ARM chips. All testing reported in this document was done on the Darwin computer cluster using an Nvidia GTX Titan X graphics card. Since ComputeCPP does not support using this card with OpenCL, ComputeCPP's experimental PTX64 bit-code generation was used to generate device code.

II Compiling

The module system on Darwin introduced complications when compiling using ComputeCPP. Since ComputeCPP is built on top of Clang and the Clang installations on Darwin are built using the gnu compilers provided by the OS (gcc 4.8), the gcc-toolchain needs to be provided to the build system. ComputeCPP does this via the COMPUTECPP_TOOLCHAIN_DIR environment variable. However, this is only used when the CMake variable CMAKE_CROSSCOMPILING is set to true. Thus, the FindComputeCpp file was edited so that the gcc-toolchain was properly set from the COMPUTECPP_TOOLCHAIN_DIR variable even when not cross compiling. Furthermore, the OpenCL::OpenCL and ComputeCPP::ComputeCpp libraries provided in FindComputeCpp.cmake were set to be global libraries as the build system could not find the libraries when the FindComputeCpp module was used by the shacl CMake build system.

Additionally, FindComputeCpp was using a non-cached CMake variable COMPUTECPP_DEVICE_COMPILER_FLAGS to propagate flags to the target specified by the add_sycl_to_target function (provided by the FindComputeCpp.cmake module). Since the module was being called within the shacl CMake build system, the non-cached variable was reading as empty by the time it was being used. To circumvent this, an interface library called ComputeCpp was added to FindComputeCpp and COMPUTECPP_DEVICE_COMPILER_FLAGS were added to the interface library's compile_options and link_libraries. After these changes, the build system worked as expected and sycl programs could be compiled from within the shacl CMake build system. The changes made to the FindComputeCpp.cmake file are detailed in Fig. 1.

Once the changes to CMake were made the ComputeCpp compiler compute++ could be used as the compiler to build SYCL code using the shacl CMake build system. The only necessary changes are to set compute++ as the CXX compiler, add the location of FindComputeCpp.cmake as a CMake module, find the ComputeCpp package, and then add SYCL to the appropriate targets, detailed in Fig. 2

```
list(APPEND COMPUTECPP_DEVICE_COMPILER_FLAGS --gcc-toolchain=${COMPUTECPP_TOOLCHAIN_DIR})
...
add_library(ComputeCpp::ComputeCpp UNKNOWN IMPORTED GLOBAL)
add_library(OpenCL::OpenCL UNKNOWN IMPORTED GLOBAL)
...
add_library(ComputeCpp INTERFACE)
target_compile_options(ComputeCpp INTERFACE "${COMPUTECPP_DEVICE_COMPILER_FLAGS}")
target_link_libraries(ComputeCpp INTERFACE "${COMPUTECPP_DEVICE_COMPILER_FLAGS}")
...
target_link_libraries(${SDK_ADD_SYCL_TARGET} PRIVATE ComputeCpp)
```

Figure 1: Changes made to FindComputeCpp.cmake module.

```
list(APPEND CMAKE_MODULE_PATH ${CMAKE_CURRENT_SOURCE_DIR}/dependencies/ComputeCpp/cmake/Modules)
find_package(ComputeCpp REQUIRED)
...
add_sycl_to_target(TARGET MGMC.${SOURCE_NAME}.SYCL.test SOURCES ${SOURCE_NAME}.SYCL.test.cpp)
```

Figure 2: Changes made to FindComputeCpp.cmake module.

III Application

III.A Basic usage of SYCL

The set up of a SYCL program feels like setting up a classic CUDA program. The SYCL runtime handles the transfer of data between the host (CPU) and device (GPU, coprocessor, etc.), however it does this through SYCL-specific data structures called buffers and accessors. The minimum amount of code to add two vectors together is shown in Fig. 3. For comparison, a program adding two vectors together using STL algorithms is shown in Fig. 4.

As seen in Fig. 3, there are several additional steps to using a SYCL kernel compared to the usual STL version. These steps include

- Create buffers from pre-allocated host data
- Create a sycl queue targeting the device you want to run on
- Create and submit a functor to the queue that
  - Takes a SYCL control group handle as an argument
  - Creates accessors to the sycl buffers
  - Creates a functor to do the parallel work that captures by value and takes a sycl id (or a sycl item) as its sole argument.
  - Calls the parallel_for method of the control group handle with the functor as the sole argument.

There are several points to note about this program:

- Buffers synchronize with the data they were created from when they go out of scope (hence the scoping brackets).
- Accessors are created with the control group handle (cgh) as an argument - they must be created from within a queue submission.
- Buffers and accessors must be created for every piece of data allocated on the heap as well as for every object that will be modified inside the functor.
- The functor or lambda submitted to parallel-for is copied to the device being executed on, and lambdas can only capture by value

```cpp
#include <CL/sycl.hpp>
#include <vector>
#include <iostream>
int main() {
  using T = double;
  size_t N = 100;
  std::vector<T> VA(N, 1.0);
  std::vector<T> VB(N, 2.0);
  std::vector<T> VC(N);
  cl::sycl::range<1> numOfItems{N};
  { // scoping curly brackets
    cl::sycl::buffer<T, 1> bufferA(VA.data(), numOfItems);
    cl::sycl::buffer<T, 1> bufferB(VB.data(), numOfItems);
    cl::sycl::buffer<T, 1> bufferC(VC.data(), numOfItems);
    auto queueFunc = [&](cl::sycl::handler& cgh) {
      auto accessorA = bufferA.template get_access<cl::sycl::access::mode::read>(cgh);
      auto accessorB = bufferB.template get_access<cl::sycl::access::mode::read>(cgh);
      auto accessorC = bufferC.template get_access<cl::sycl::access::mode::read_write>(cgh);
    auto parallelFunc = [=] (cl::sycl::id<1> wiID){
        accessorC[wiID] = accessorA[wiID] + accessorB[wiID];
      };
      cgh.parallel_for<class SimpleVadd>(numOfItems, parallelFunc);
    };
    cl::sycl::queue deviceQueue(cl::sycl::gpu_selector{});
    deviceQueue.submit(queueFunc);
  } // scoping curly brackets
  for (auto& val : VC) std::cout << val << "\n";
  return 0;
}
```

Figure 3: SYCL program to add two vectors together.

```cpp
#include <vector>
#include <algorithm>
int main() {
  int N = 100;
  std::vector<T> VA(N, 1.0);
  std::vector<T> VB(N, 2.0);
  std::vector<T> VC(N);
  auto func = [](const double& A, const double& B){return A + B;}
  std::transform(VA.begin(), VA.end(), VB.begin(), VC.begin(), func);
  for (auto& val : VC) std::cout << val << "\n";
  return 0;
}
```

Figure 4: C++ program to add two vectors together.

```
#include <vector>
#include <sycl/execution_policy>
int main() {
  class A { public: double a = 0.0; };
  class B { public: int b = 0.0; };
  class C : public A, public B { };

  size_t N = 2000;
  std::vector<C> data(N);
  using Buffer = cl::sycl::buffer<C, 1>;
  using Range = cl::sycl::range<1>;
  Buffer buf(data.data(), Range(N));
  cl::sycl::queue q(cl::sycl::host_selector{});
  q.submit([&](cl::sycl::handler &cgh) {
    auto acc = buf.template get_access<cl::sycl::access::mode::read_write>(cgh);
    cgh.parallel_for<class ForEach>(Range(N), [=](cl::sycl::item<1> item) {
      auto id = item.get_linear_id();
      acc[id].a = id*0.6;
      acc[id].b = id;
    });
  });
}
```

Figure 5: SYCL program that fails to compile due to C not being a standard layout type.

Additionally, the functor being passed to parallel-for must adhere to SYCL-specific requirements. Some of these requirements are common among all (as far as the author is aware) heterogeneous programming paradigms, with the main requirement being that data allocated on the host cannot be directly used on the device without some form of data management, whether it's explicit memory copies (cudaMemcpy), use of memory management objects (sycl::buffer), or behind-the-scenes run-time data management (CUDA managed memory). However, some requirements are language-specific. One such case is the requirements specified by the SYCL standard in regards to the kernel (functor being passed to parallel-for) [3]:

- C++ standard layout values must be passed by value to the kernel.
- C++ non-standard layout values must not be passed as arguments to a kernel that is compiled for a device.

A standard layout type has a long list of requirements from the standard (cppref), with the most relevant points being:

- Has no virtual functions or virtual base classes
- Has no non-static data members of reference type
- Has no base classes with non-static data members, or has no non-static data members in the most derived class and at most one base class with non-static data members

The first two requirements line up with the usual requirement of explicit memory management and are common among both CUDA and SYCL kernels. References initialized in host code and passed to device code would still point to host memory, and virtual functions use pointers to a vtable determine which function to call, and that vtable is located in host memory. However, the third requirement is not necessary from a memory management perspective and is not a requirement when using CUDA. This third requirement prohibits the SYCL code in Fig. 5 from compiling the simple code example shown in Fig. 5.

Since class C inherits from classes A and B and both A and B contain non-static data members, class C is not a standard layout class and thus the code fails to compile, throwing the error that class C is not standard layout and thus cannot be accessed from within a SYCL kernel. This requirement to only use standard-layout classes in SYCL kernels is limiting from a design perspective, as it prohibits the facet design pattern from being used within any device code. For example, the shacl::particle library, which is intended to have a common set of particle methods shared between Monte Carlo codes, cannot be used as intended within SYCL projects. For the purposes of this project, a new SYCLParticle class was created to circumvent this issue, at the cost of duplicated code.

```cpp
#include <vector>
#include <iostream>
#include <sycl/execution_policy>
#include <experimental/algorithm>
using namespace std::experimental::parallel;
int main() {
  using T = double;
  int N = 100;
  std::vector<T> VA(N, 1.0);
  std::vector<T> VB(N, 2.0);
  std::vector<T> VC(N);
  auto func = [](const double& A, const double& B){return A + B;};
  sycl::sycl_execution_policy<class transform1> sepn1;
  transform(sepn1, VA.begin(), VA.end(), VB.begin(), VC.begin(), func);
  for (auto& val : VC) std::cout << val << "\n";
  return 0;
}
```

Figure 6: SYCL parallel STL program to add two vectors together.

III.B SYCL Parallel STL

The SyclParallelSTL [4] can be used to simplify the use of SYCL in a way that conforms to the standard library's STL algorithm API without requiring the user to construct SYCL buffers or accessors. The same vector addition algorithm is demonstrated in Fig. 6 using SyclParallelSTL. The creation of buffers, queues, and accessors is being handled within the SyclParalellSTL library. The only additional argument that is required in the SYCL STL version in Fig. 6 versus the STL version in Fig. 4 is the SYCL execution policy, which is passed as the first argument in the experimental version of the STL algorithm to determine how to parallelize the algorithm. However, if this program is targeting the GPU then this approach will synchronize the sycl buffers with the host vectors every time this algorithm is being called, i.e. the VA, VB, and VC vector data will be copied into and out of GPU memory every time transform is called. To avoid this, sycl buffers can be explicitly created and iterators to buffers passed into transform instead of iterators to STL containers, demonstrated in Fig. 7. Again, since buffers are now explicitly used, scoping must be used to synchronize buffer data with the original vector data.

III.C Attempted Work-arounds

The need to use buffers and accessors adds verbosity to the code, limits design choices, and is language specific. SYCL code cannot be compiled without a SYCL compiler, and the use of buffers, accessors, queues, etc, means it's difficult to abstract away anything but basic operations on POD types. The goal is to be able to use containers or classes composed of containers within SYCL kernels to access heap-allocated data like cross section or geometry data. Currently, if one needs to use heap-allocated data within a SYCL kernel then the data must be passed to buffer and accessed via accessors. If buffers and accessors could be abstracted away into container classes then SYCL code could look a lot like standard C++ with the addition of a few initialization function calls. Attempts in this work to abstract away accessors and buffers include

1. Creating containers that use accessors and buffers for data management
2. Creating view-like classes that use accessors to access buffer data managed elsewhere
3. Multiple parallel_for calls per queue submission

Attempt 1 fails since buffers do not meet the requirements of a standard layout type, and thus cannot be used in kernels or in objects contained within kernels. Attempt 2 fails with the exception cl::sycl::feature_not_supported because accessors cannot be used in a control group's kernel without being initialized using that control group (even if that control group was spawned from the same queue) in ComputeCpp. That is, accessors must be initialized using the queue's control group handle every time they are used in a kernel submitted to a queue. Finally, attempt 3 fails because launching two parallel_for calls within the same functor submitted to a queue results in a sycl exception. The code used to test attempt 2 is shown in Fig. 8. While buffers cannot be data members of objects in SYCL kernels, raw pointers to buffers can be used. Attempts to use smart points in device code result in compilation errors since their destructors call the delete method, and an objects destructor is called within SYCL a kernel, and this is not allowed by the SYCL standard. Thus, a combination of accessors and pointers to buffers could be used to create

```cpp
#include <vector>
#include <iostream>
#include <sycl/execution_policy>
#include <experimental/algorithm>
#include <sycl/helpers/sycl_buffers.hpp>
using namespace std::experimental::parallel;
using namespace sycl::helpers;
int main() {
  int N = 1000;
  using T = double;
  std::vector<T> VA(N, 1.0);
  std::vector<T> VB(N, 2.0);
  std::vector<T> VC(N);
  { // scoping brackets
    cl::sycl::buffer<T, 1> bufferA(VA.begin(), VA.end());
    cl::sycl::buffer<T, 1> bufferB(VB.begin(), VB.end());
    cl::sycl::buffer<T, 1> bufferC(VC.begin(), VC.end());
    sycl::sycl_execution_policy<class transform1> sepn1;
    auto func = [](const double& A, const double& B){return A + B;};
    for (int j = 0; j < 100; j++){
      transform(sepn1, begin(bufferA), end(bufferA), begin(bufferB), begin(bufferC), func);
    }
  } // scoping brackets
  for (auto& val : VC) std::cout << val << "\n";
  return 0;
}
```

Figure 7: SYCL parallel STL program to add two vectors together demonstrating use of buffers to prevent unnecessary copies to/from the device.

a container “view” which could be used in SYCL kernels, however the view still needs to be initialized with a queue’s control group handler for every queue submission. Furthermore, destructing the buffer is no longer sufficient to sync the buffer’s data with the original vector – the accessor must also go out of scope for this to occur. Thus, it is best to have the accessed buffer have the same lifetime as the accessor and make use of host_buffer accessors to view the data on the host. This means that range-based for loops are no longer an option, as accessors do not have begin and end methods. Thus, a raw for loop and a host accessor is used to view the data after the kernel submission. An example of this code is shown in Fig. 9. While this AccessHolder enables the use of accessors within classes, they still need to be initialized on every queue submission, and the buffers need to exist somewhere, so the utility of this is somewhat questionable.

III.D SYCL for Monte Carlo Transport

While the use of SYCL for basic operations on simple data is straightforward, using SYCL for more advanced applications becomes cumbersome and compiler-error prone. The most apparent difficulty encountered in this work is the need to use SYCL buffers and accessors to access heap-allocated data on the device. While this was not a major burden for operating on vectors of doubles, it becomes a difficulty when attempting to use heap-allocated data within the kernel. Since the kernel cannot be passed by reference to the transform algorithm, it should be light-weight. This precludes having memory-intensive data members directly inside the functor, like cross-section information and geometry information. Furthermore, the functor must contain only data members that are standard layout types; it must capture all data by value, none of the data can have virtual base classes, and none of the data can have more than one base class containing non-static data members.

These requirements combine to make SYCL difficult to use for a transport application. Figure 10 shows the main transport kernel written using SYCL, and Fig. 11 shows the same kernel written for OpenMP and CUDA.

The primary issue with the SYCL kernel in Fig. 10 is that it is not compatible with the SyclParallelSTL API since it requires additional buffers and accessors beyond just the data (source bank) that is fed into the algorithm via the API (see Fig. 11). While the SYCL buffers would realistically be initialized ahead of time at the beginning of the simulation, the accessors cannot be constructed ahead of time since they require the control group handle (cgh) and that is only present within the queue submission function call. Furthermore, buffers do not meet the requirements to be considered a standard layout type (they contain virtual destructors, at least), so they cannot be data members of kernels or data members of objects within kernels. Thus, the buffers and accessors cannot be hidden within containers that would ordinarily hold data and all methods necessary to access said data. As such, it is impossible to directly

```
#include <vector>
#include <iostream>
#include <CL/sycl.hpp>
template <typename T>
class AccessorHolder{
  public:
  using accessor_t = cl::sycl::accessor<T, 1, cl::sycl::access::mode::read_write, cl::sycl::access
    ::target::global_buffer, cl::sycl::access::placeholder::true_t>;
  accessor_t acc;
  T& operator[] (size_t index) const noexcept { return acc[index]; }
};

int main() {
  using T = double;
  using Buffer = cl::sycl::buffer<T, 1>;
  using Range = cl::sycl::range<1>;
  int N = 10;
  std::vector<T> VA(N, 1.0);
  Range numOfItems(N);
  {
    AccessorHolder<T> accA;
    cl::sycl::queue q(cl::sycl::host_selector{});
    {
      Buffer bufferA(VA.data(), numOfItems);
      q.submit([&](cl::sycl::handler& cgh) {
        accA.acc = decltype(accA.acc)(bufferA, cgh);
        cgh.parallel_for<class AccA>(numOfItems, [=](cl::sycl::item<1> item){
          auto id = item.get_linear_id();
          accA[id] = 5.0;
        });

        cgh.parallel_for<class AccA2>(numOfItems, [=](cl::sycl::item<1> item){
          auto id = item.get_linear_id();
          accA[id] = 10.0;
        });
      });

    }
  }
  std::cout << " PRINTING A \n";
  for (auto& val : VA) std::cout << val << "\n";
  return 0;
}
```

Figure 8: Unsuccessful attempt to use accessors that are initialized once in multiple queue submissions.

```cpp
#include <vector>
#include <iostream>
#include <memory>
#include <CL/sycl.hpp>

template <typename T>
class AccessorHolder{
  private:
  using accessor_t = cl::sycl::accessor<T, 1, cl::sycl::access::mode::read_write, cl::sycl::access
    ::target::global_buffer, cl::sycl::access::placeholder::true_t>;
  using host_accessor_t = cl::sycl::accessor<T, 1, cl::sycl::access::mode::read_write, cl::sycl::
    access::target::host_buffer>;
  using Buffer = cl::sycl::buffer<T, 1>;
  Buffer* bufferPtr_;
  accessor_t acc_;
  public:

  AccessorHolder(Buffer* bufferPtr): bufferPtr_(bufferPtr) { }

  void initialize(cl::sycl::handler& cgh){
    acc_ = accessor_t(*bufferPtr_, cgh);
  }

  T& operator[] (size_t index) const noexcept { return acc_[index]; }
  host_accessor_t get_host_accessor() { return {*bufferPtr_}; }
};

int main() {
  using T = double;
  using Buffer = cl::sycl::buffer<T, 1>;
  using Range = cl::sycl::range<1>;
  int N = 10;
  std::vector<T> VA(N, 1.0);
  Range numOfItems(N);
  Buffer bufferA(VA.data(), numOfItems);
  AccessorHolder<T> accA(&bufferA);
  {
    cl::sycl::queue q(cl::sycl::host_selector{});
    {
      q.submit([&](cl::sycl::handler& cgh) {
        accA.initialize(cgh);
        cgh.parallel_for<class AccA>(numOfItems, [=](cl::sycl::item<1> item){
          auto id = item.get_linear_id();
          accA[id] = 5.0;
        });
      });
    }
  }
  std::cout << " PRINTING A \n";
  auto h_acc = accA.get_host_accessor();
  for (int i = 0; i < N; i++){
    std::cout << h_acc[i] << "\n";
  }
  return 0;
}
```

Figure 9: Successful attempt to use buffer pointers and accessors in an object passed to a kernel.

```
using ParticleBuffer = cl::sycl::buffer<Particle, 1>;
using Range = cl::sycl::range<1>;
using NextIDBuffer = cl::sycl::buffer<ExecPolicy::AtomicIndex_t, 1>;
using XSBuffer = cl::sycl::buffer<MGXS, 1>;
using EigenvalueTallyBuffer = cl::sycl::buffer<EigenvalueTally_t, 1>;
cl::sycl::queue q(cl::sycl::gpu_selector{});
{ // Scoping brackets
  ParticleBuffer sourceBankBuffer(sourceBank->data(), Range(3*nParticles));
  ParticleBuffer fissionBankBuffer(fissionBank->data(), Range(3*nParticles));
  NextIDBuffer fissionBankNextIDBuffer(&fissionBank->getNextID(), Range(1));
  XSBuffer xsBuffer(xsPtr, Range(1));
  EigenvalueTallyBuffer eigenvalueTallyBuffer(eigenvalueTallyPtr, Range(1));
  q.submit([&](cl::sycl::handler &cgh){
    auto sourceBankAcc = sourceBankBuffer.template get_access<cl::sycl::access::mode::read_write>(
    cgh);
    auto nextIDAcc = fissionBankNextIDBuffer.template get_access<cl::sycl::access::mode::read_write
    >(cgh);
    auto fissionBankAcc = fissionBankBuffer.template get_access<cl::sycl::access::mode::read_write
    >(cgh);
    auto xsAcc = xsBuffer.template get_access<cl::sycl::access::mode::read_write>(cgh);
    auto eigenvalueTallyAcc = eigenvalueTallyBuffer.template get_access<cl::sycl::access::mode::
    read_write>(cgh);
    auto transportFunctor = [=] (cl::sycl::item<1> history){
      auto historyID = history.get_linear_id();
      Particle& p = sourceBankAcc[historyID];
      auto& xs = xsAcc[0];
      auto& eigenvalueTally = eigenvalueTallyAcc[0];
      auto& nextID = nextIDAcc[0];
      while (p.alive()) {
        auto dist = getDistanceAndMoveParticle(p, xs, eigenvalue);
        eigenvalueTally.tallyTrack(dist, p, xs);
        if (p.dead()) {
          eigenvalueTally.tallyLeakage(p);
        } else {
          // Process collision - copied and pasted here due to ComputeCPP compiler bug
          auto& bank = fissionBankAcc;
          auto rx = sampleReaction(p, xs, eigenvalue, mode);
          int neutrons = generateNeutrons(p, xs, eigenvalue, mode);
          for ( int n = 0 ; n < neutrons ; n++ ) {
            Real outgoingAngle = isotropicPolarDistribution.sample(p.rng());
            int outgoingGroup = xs.sampleFissionSpectrum(p.rng());
            int id = nextID++;
            bank[id] = p;
            bank[id].u(outgoingAngle);
            bank[id].g(outgoingGroup);
          }
          if (rx == Reaction::Scattering) {
            scatter(p, xs, p.rng());
          } else if (rx == Reaction::AlphaAbsorption) {
            if (eigenvalue < 0) { // alpha time source
              bank[nextID++] = p;
            } else { // alpha absorption
              p.kill();
            }
          } else { // absorption
            p.kill();
          }
        }
      }
    };
    cgh.parallel_for<class TransportFunctor>(Range(sourceBank->size()), transportFunctor);
  });
} // scoping brackets synchronize buffer data
```

Figure 10: SYCL transport kernel

```

auto transportFunctor = [=] DEVICE (Particle& p) {
  auto& xs = *xsPtr;
  while (p.alive()) {
    auto dist = getDistanceAndMoveParticle(p, xs, eigenvalue);
    eigenvalueTallyPtr->tallyTrack(dist, p, xs);
    if (p.dead()) {
      eigenvalueTallyPtr->tallyLeakage(p);
    } else {
      processCollision(p, xs, *fissionBankPtr, eigenvalue);
    }
  }
};

for_each(sourceBank->begin(), sourceBank->end(), transportFunctor, execPolicy);
```

Figure 11: Serial/OpenMP/CUDA transport kernel

use containers that allocate and access data on the heap within a SYCL kernel; the container data must be passed to a buffer, and then only accessed from an accessor created from that buffer within the queue submission function call. While it is possible to remove several of the buffers and accessors in the SYCL kernel in Fig. 10 by making the transport kernel adhere to a more functional programming style, the cross section and geometry information would still need buffers and accessors in a realistic problem.

Additionally, it should be noted that the SYCL kernel has approximately 23 additional lines of code due to a compiler bug that didn't like the processCollision functor that was used in the code in Fig. 11, so the functor code was copy-and-pasted into the transport kernel. Furthermore, it should be noted that the CUDA and OpenMP versions of the code in Fig. 11 require supporting code in the form of language-specific data structures and compiler macros. However, this additional supporting code can be used at a higher level, and does not require interfering with the low-level transport routines as seen in the SYCL implementation in Fig. 10.

The SYCL transport kernel compiles and runs on the host, producing the same output as the serial version of the code in Fig. 11. However, initializing the queue with the gpu_selector results in a series of ComputeCpp warnings: " [Computecpp:CC0035]: Intrinsic llvm.fma.f64 has been generated in function SYCL_class_TransportFunctor which is illegal in SPIR and may result in a compilation failure [-Wsycl-undef-func]". The program compiles, but when it is run it produces the exception: cl::sycl::compile_program_error, indicating that the fused-multiply-add instructions are causing issues. Attempts to disable fma by compiling with the -mno-fma flag have failed; the flag is accepted but the warnings still appear and program behavior is unchanged. Additionally, the cross-section information contained within xsAcc would need to be specified as fixed-length arrays as used in Fig. 10, otherwise additional buffers and accessors would need to be created for every cross-section, fission, total, scatter, etc.

A possible way to abstract away the use of buffers and accessors would be to create a Buffer and Accessor class for every class that uses heap-allocated data. For example, one could create a XSBuffer class that contains buffers for every XS type (total, fission, etc.), and a XSAccessor class that contains all the necessary SYCL accessors and methods to access the underlying data. The XSAccessor would be constructed at the start of every queue submission using a XSBuffer and a control group handle. This was not attempted in this work as it is the author's opinion that sufficient evidence is presented to discourage the use of SYCL until a computer system that requires OpenCL for parallelization is planned for construction at the lab.

IV Conclusions

SYCL is promising from a perspective of parallelization on all types of heterogenous computing systems ranging from AMD GPUs, Nvidia GPUs, Intel graphics technology, and potentially ARMs chips, however it is apparent that the cost for using SYCL includes the addition of invasive and verbose code in the low-level portions of the code, limiting restrictions on class definitions, and occasional compiler bugs. It is the author's opinion that SYCL is not worth the effort until a computer system exists in the national lab system that requires OpenCL for parallelization.

References

[1] "SYCL," Https://www.khronos.org/sycl/ (2019).

[2] “ComputeCpp,” Https://www.codeplay.com/products/computesuite/computecpp (2019).

[3] “SYCL Specification,” Https://www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf (2019).

[4] “SyclParallelSTL,” Https://github.com/KhronosGroup/SyclParallelSTL (2019).